PEPSI: Practically Efficient Private Set Intersection in the Unbalanced Setting

Rasoul Akhavan Mahdavi, Nils Lukas, Faezeh Ebrahimianghazani, and Thomas Humphries, *University of Waterloo;* Bailey Kacsmar, *University of Alberta;* John Premkumar and Xinda Li, *University of Waterloo;* Simon Oya, *University of British Columbia;* Ehsan Amjadian, *University of Waterloo and Royal Bank of Canada;* Florian Kerschbaum, *University of Waterloo*

https://www.usenix.org/conference/usenixsecurity24/presentation/mahdavi

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

PEPSI: Practically Efficient Private Set Intersection in the Unbalanced Setting

Rasoul Akhavan Mahdavi[1], Nils Lukas[1], Faezeh Ebrahimianghazani[1], Thomas Humphries[1], Bailey Kacsmar[2], John Premkumar[1], Xinda Li[1], Simon Oya[3], Ehsan Amjadian[1,4], and Florian Kerschbaum[1]

[1]University of Waterloo
[2]University of Alberta
[3]University of British Columbia
[4]Royal Bank of Canada
{rasoul.akhavan.mahdavi, nlukas, f5ebrahi, thomas.humphries, jpremkumar, xinda.li, ehsan.amjadian, florian.kerschbaum}@uwaterloo.ca, kacsmar@ualberta.ca, simon.oya@ubc.ca

Abstract

Two parties with private data sets can find shared elements using a Private Set Intersection (PSI) protocol without revealing any information beyond the intersection. Circuit PSI protocols privately compute an arbitrary function of the intersection - such as its cardinality, and are often employed in an *unbalanced* setting where one party has more data than the other. Existing protocols are either computationally inefficient or require extensive server-client communication on the order of the larger set. We introduce Practically Efficient PSI or PEPSI, a non-interactive solution where only the client sends its encrypted data. PEPSI can process an intersection of 1024 client items with a million server items in under a second, using less than 5 MB of communication. Our work is over 4 orders of magnitude faster than an existing non-interactive circuit PSI protocol and requires only 10% of the communication. It is also up to 20 times faster than the work of Ion et al., which computes a limited set of functions and has communication costs proportional to the larger set. Our work is the first to demonstrate that non-interactive circuit PSI can be practically applied in an unbalanced setting.

1 Introduction

Privacy-preserving data analytics operates on the principles that (i) data is accessible and can be analyzed while (ii) ensuring that the user's privacy remains uncompromised. For instance, consider the application of secure contact discovery used in private messengers such as Signal [16, 24]. Users want to connect with their friends without sharing their entire contact list with the server, and the server does not want to disclose the list of all contacts to each user. The revelation of this information by the client (or server) would defeat the very essence of these services.

Private Set Intersection (PSI) protocols offer a solution to this problem. PSI is designed to compute the intersection of two sets while ensuring that nothing beyond the intersection is revealed. A specific form of PSI is *unbalanced* PSI, in which one party, typically the client, has a substantially smaller set than the second party, the server. Moreover, the client is often a resource-constrained device like a mobile phone or an IoT device with limited network connectivity. Consequently, it benefits the client by minimizing network communication, reducing the number of network round trips, and offloading computational tasks to the server.

In some cases, the goal of the client and server is to compute functions of the intersection, rather than to learn the intersection itself. For instance, in the case of COVID-19 exposure applications, a user simply wants to know if they share any randomly generated codes with those stored on the server, indicating potential exposure [17, 36, 37, 39]; who those matches are is irrelevant. Similarly, when gauging the effectiveness of ad campaigns by cross-referencing online advertisements with offline credit card transactions [22], the advertiser is interested in knowing the total sales corresponding to a specific set of credit cards used at certain vendors. Only the total sales amount is of interest, not the individual transactions. *Circuit PSI* is the term used when a function of the intersection is calculated rather than the intersection itself.

In the circuit PSI setting, the work of Kacsmar et al. [23] (referred to as DiPSI) is best suited for the unbalanced non-interactive setting. However, DiPSI still suffers from inefficiency issues. In contrast to DiPSI, the numerous fast state-of-the-art solutions for PSI [7, 14, 34] are limited in terms of being extendable to the circuit PSI setting. Specifically, each state-of-the-art solution is limited in one of the following ways. The first limitation is for solutions that are restricted as to the functions they can compute [22, 28]. They are unable to execute functions beyond a specific predefined set. Despite the fact that protocols designed for specific functions allow targeted optimizations, our evaluation shows that we outperform the state-of-the-art for many specific functions, particularly for large server set sizes. Furthermore, targeted optimizations become obsolete when the function needs to be changed. Second, adapting some solutions to circuit PSI levies a computational and communication burden on the client to compute arbitrary functions, usually in the form of

Table 1: Comparing the properties of different PSI protocols. m and n are the client and server set sizes, respectively. We assume $m \ll n$ in the analysis of works which support the unbalanced setting. DH: Diffie-Hellman. 2PC: two-party computation. PDTE: private decision tree evaluation. *PSI using [22, 30, 42] is done in one round and an extra round is required to compute the sum. Moreover, it can only perform a limited set of functions in only one extra round. **PSI is realized by extending private set membership.

Work	Tools	Unbalanced	Rounds	Comm.	Comp.	PSI	Labelled PSI	Circuit PSI
[35]	B-OPPRF + 2PC	×	$1+\log_2 \lambda$	$O(n)$	$O(n \log n)$	✓	✓	2PC
[7]	RB-OPPRF + 2PC	×	$1+\log_2 \lambda$	$O(n)$	$O(n)$	✓	✓	2PC
[22, 42]	DH	×	1+1 *	$O(n)$	$O(n)$	✓	×	✓*
[13, 30]	OT + PDTE	×	1+1	$O(n)$	$O(n)$	✓	✓**	✓**
[11]	HE	✓	1	$O(m \log n)$	$O(n \log n)$	✓	✓	×
[10, 14]	OPRF + HE	✓	2	$O(m)$	$O(n \log n)$	✓	✓	×
DiPSI [23]	HE	✓	1	$O(m)$	$O(n)$	✓	✓	✓
PEPSI	HE	✓	1	$O(m)$	$O(n)$	✓	✓	✓

two-party computation [7, 33]. This communication burden is usually proportional to the larger set [7, 22, 33] or depends on the complexity of the task. Such a burden is not acceptable when designing a non-interactive protocol.

Summary of Contributions. In this work, we propose PEPSI, an efficient, non-interactive circuit PSI protocol using homomorphic encryption. Homomorphic Encryption (HE) is a form of encryption that permits computation on the data while in encrypted form. In PEPSI, the client encrypts its elements using an HE scheme and sends them to the server, which compares the elements homomorphically. The output of each comparison is binary and is used to compute functions. Our approach removes the limitations that are present in existing work. PEPSI does not restrict the function that can be evaluated, there is no communication or computation burden on the client to compute the function, and it only has communication complexity proportional to the smaller client set size.

Our approach also overcomes the impractical runtimes associated with the DiPSI protocol from the literature [23]. We address the slow comparisons by comparing items using the efficient constant-weight equality operator [19], permutation-based hashing, and cuckoo hashing. Altogether this reduces the size of the elements that need to be compared such that PEPSI is over four orders of magnitude faster than DiPSI and requires 90% less communication. For example, PEPSI can compute the intersection of 1024 and one million 32-bit elements in under one second with less than 5 MB of communication.

PEPSI has a runtime and communication cost comparable to other PSI protocols based on HE [10, 14], which cannot extend to circuit PSI. Moreover, the client is required to do much less computational work in PEPSI, making it compatible with use cases where the client does not have much computational power, i.e., the unbalanced setting. We empirically compare PEPSI with state-of-the-art protocols in our experiments, showing that PEPSI is even competitive with PSI protocols which cannot extend to circuit PSI. Our work is the first to show that non-interactive circuit PSI can be efficient up to the point it is practical for use in applications.

2 Related Work

The term *private set intersection* (PSI) was coined by Freedman et al. [21]. In a PSI protocol, two parties compute the intersection of their respective private sets such that no information is revealed except the intersection itself. If the intersection is only revealed to one of the parties, this is known as one-sided PSI. However, one-sided PSI can be extended to two-sided PSI by running the protocol a second time and switching the roles of the client and server.

2.1 PSI with Leakage

In some applications and real-world use cases, such as checking for compromised credentials, the overhead of PSI is high if the client element is compared to all elements. For example, comparing the client credentials to billions of leaked credentials is expensive. To alleviate the overhead, some works propose to only compare the client elements to a subset of the server elements [29, 38]. For example, elements are distributed into buckets by their prefix. To query for a specific element, the client only queries the bucket corresponding to that element instead of the entire set. The server will know that the client's query falls into that specific bucket so such an approach partially leaks information about the client at the cost of better performance. This leakage degrades the security guarantees but may be acceptable in some applications. Thomas et al. [38] predict that the prefix of a user's creden-

tials is common with about 350k-470k other users, assuming each client has one credential in the database.

Our work does not follow this approach and instead offers PSI without leakage. While such an approach requires more computational effort, it offers better security. We note that our protocol could also be combined with bucketization, offering a tunable trade-off between security and performance from the leakage of current protocols [29, 38] to no leakage.

2.2 Unbalanced PSI

Unbalanced PSI is a special case of PSI where one party, which we call the client, has a much smaller set compared to the other party, i.e., the server [1, 2, 10, 11, 14, 17, 24, 29]. This is a common assumption in many applications of PSI. For example, in compromised credentials checking (C3) [24, 29], the client wishes to access a large database containing credentials that are leaked on the web. While the set of client credentials is small, e.g., a few hundred credentials, the database of leaked credentials is in the billions and growing [24, 29]. Moreover, the client is usually bandwidth-constrained and has limited computational power. Hence, protocols with limited network round trips and little client-side computation are preferred. A narrower variant of unbalanced PSI exists, dubbed laconic PSI, with the additional constraint that only the server learns the output of the protocol [1, 2].

While there are many efficient state-of-the-art PSI protocols [7, 30, 32, 34, 35], the communication complexity of these solutions scales with the larger set, making them unsuitable for unbalanced PSI. For example, Pinkas et al. and Chandran et al. propose PSI protocols that compute an oblivious PRF for each element of the server set [7, 35]. Ma and Chow propose to construct a private decision tree from the server set, which is encrypted and sent to the client. The client then obliviously traverses this decision tree using OT [30]. Such solutions are useful in high-bandwidth, low-latency networks but fall short over high-latency networks.

Amongst solutions that have no leakage, the current most efficient non-interactive PSI protocols in the unbalanced case are based on homomorphic encryption [10, 11, 14]. Homomorphic encryption (HE) is a type of encryption that allows computations on data in its encrypted form. Chen et al. [11] were among the first to propose a PSI protocol specifically designed for the unbalanced setting using homomorphic encryption. If $\mathbb{Y}$ represents the server set, the key idea is to construct the polynomial $P(x) = r\prod_{y_i \in \mathbb{Y}}(x - y_i)$, for some random value r. The client sends an encryption of its elements to the server, which then homomorphically evaluates $P(x)$ on the client input. For a certain client element x_0, if $x_0 \in \mathbb{Y}$, then $P(x_0) = 0$, otherwise $P(x_0)$ is a random number. The idea of representing the set as a polynomial had been proposed before [26] and Chen et al. included optimizations to reduce the multiplicative depth of the function that is homomorphically evaluated, thereby enhancing efficiency [11]. Chen et al. [10] and Cong et al. [14] built upon [11], employing a combination of Oblivious Pseudorandom Functions (OPRF) and homomorphic encryption. This improved performance extended the security to the malicious model and also allowed the protocol to extend to elements of arbitrary bitlength.

2.3 Labelled PSI

A specific variant of PSI is *labelled PSI*, where the server has a private label associated with each element in its private set. The server stores pairs in the form of (y_i, ℓ_i) where y_i is the identifier of the element and ℓ_i is the label associated with that element. The two parties compute the intersection of their sets and output all the pairs (y_i, ℓ_i) where y_i is in the intersection of the two sets. Nothing beyond the elements in the intersection and their corresponding labels are revealed to either one of the parties. When the client has only one element, this problem is equivalent to private information retrieval by keywords, first proposed by Chor et al. [12]. In the context of private contact discovery, the client may wish to retrieve the public keys of users in their contact list [16, 24].

Many PSI protocols can be extended to labelled PSI with additional computation and communication costs. For example, Chen et al. [10] and Cong et al. [14] extend their protocol to labelled PSI by evaluating another polynomial which evaluates to the ℓ_i when $x = y_i$. For the protocols of Pinkas et al. [33] and Chandran et al. [7] the extension to labelled PSI can be done without changing the asymptotic complexity, but it does result in doubling the concrete communication and computation cost.

2.4 Circuit PSI

Another variant of PSI is *circuit PSI*, where the objective is to compute a function of the intersection, rather than the intersection itself. One example is PSI-Cardinality, where the parties compute the size of the intersection [23]. Another example is PSI-Sum [22], where the sum of values associated with elements in the intersection is revealed.

One application for PSI-Sum, known as ad-conversion, is assessing the effectiveness of ad campaigns [22]. A company purchasing an ad through an ad service company, such as Google, would like to know the total purchases made by users who have seen their particular ad. If Google can show a client who paid to have an ad displayed through their system that the ad leads to sales, then it is easier to convince the client to purchase more ads in the future. Such ad-conversion computations can be done by linking those who have seen an ad with their credit card transactions. However, credit card vendors are unwilling to disclose their clients' transactions. PSI-Sum is a solution in this scenario.

In the existing approaches [10, 11, 14], every comparison between a client and server element results in an arithmetic

output. An arithmetic output means that if the elements are equal, the output is zero, and the output is a random non-zero number in $\mathbb{Z}_p$, for some p. While using this approach results in fast PSI protocols, computing a function of the intersection is not feasible. To compute a function, the result of each comparison must be converted to a binary output, i.e., one for a match and zero otherwise.

One approach to computing such functions is to use secure two-party computation (2PC) to convert the arithmetic output into a secret-shared binary output. 2PC can then be used to compute any arbitrary function over the binary output. While there is flexibility regarding the functions that can be computed in this approach, the communication complexity of MPC protocols is high. More specifically, the communication complexity depends on the size of both sets and the complexity of the desired function [7, 33].

Another approach is to compare elements from the client and server homomorphically via a homomorphic equality circuit. The output of the comparisons is binary in this case. This is the only non-interactive approach in the literature and is more suitable for the unbalanced setting since it does not require any client interaction. The work of Kacsmar et al. [23], which they call DiPSI, uses this approach. DiPSI achieves asymptotically optimal communication, given that only the client dataset is communicated over the network. The authors use this protocol to compute a differentially private PSI-Sum and other functions. The main problem with DiPSI is the extremely high runtime. PSI-Sum between 1024 client elements and 1 million server elements can take over 600 minutes. This can mainly be attributed to the homomorphic comparison function that they employ. We propose a solution that compares elements homomorphically, similar to DiPSI but is over four orders of magnitude faster.

Some works take a different approach to circuit PSI by designing a protocol that can compute a very specific function very efficiently [22, 28]. This is in contrast to the other approaches where an arbitrary function can be derived. A specialized protocol outperforms generalized protocols in terms of performance but restricts the functions that can be derived and may leak additional information in the process [22].

Ion et al. [22] propose a protocol for PSI-Sum-with-Cardinality. This protocol outputs, to the server, the sum of values associated with elements in the intersection, whilst also revealing the intersection cardinality to the client. Lepoint et al. [28] proposed Private Join and Compute (PJC) which provides PSI-sum, which is the sum of values associated with elements in the intersection, without leakage of the intersection cardinality. However, their protocol cannot compute any function other than the sum and has an expensive offline phase. PSI-Stats follows a similar approach and supports more functions including arithmetic and geometric mean, standard deviation, and approximate composition [42]. However, this protocol is still limited to the functionalities proposed by the authors. A common feature of these works is they require three rounds of interaction and the final results are revealed to the server.

3 Background

3.1 The FV Cryptosystem

The FV cryptosystem [18] permits computation over vectors of numbers using Single Instruction, Multiple Data (SIMD) operators. Plaintexts are vectors of length N where each element is in $\mathbb{Z}_t$. N and t are called the *polynomial modulus degree* and the *plaintext modulus*, respectively. Ciphertexts are vectors of polynomials with coefficients in $\mathbb{Z}_q$, where q is called the *ciphertext modulus* and $\mathcal{C}$ is the ciphertext space. The following operations are permitted in FV:

- Addition: Given $c_X, c_Y \in \mathcal{C}$ which encrypts $X, Y \in \mathbb{Z}_t^N$, respectively, output c_A which encrypts $X+Y$.
- Plain Multiplication: Given $X \in \mathbb{Z}_t^N$ and $c_Y \in \mathcal{C}$ which encrypts $Y \in \mathbb{Z}_t^N$, output c_{PM} which encrypts $X \odot Y$.
- Multiplication: Given $c_X, c_Y \in \mathcal{C}$ which encrypt $X, Y \in \mathbb{Z}_t^N$, respectively, output c_M which encrypts $X \odot Y$.

Note the addition ($+$) and multiplication ($\odot$) operations from FV over vectors in $\mathbb{Z}_t$ are defined element-wise.

3.2 Constant-weight Encoding

A constant-weight code is a collection of binary codewords with the same Hamming weight, i.e., each codeword has a fixed number of bits set to one. Throughout this paper, we denote the common Hamming weight that the codewords share as h. We denote the set of ℓ-bit constant-weight codewords with Hamming weight h by $CW(\ell, h)$.

We use the arithmetic constant-weight equality operator from Mahdavi and Kerschbaum [19] for comparing constant-weight codewords with Hamming weight h. The choice of h impacts the computation cost of the operator and the required communication. This operator does not depend on the operands which it is comparing, and so can be computed using SIMD operations. Algorithm 1 describes this operator. Throughout this paper, we denote $[n] = \{0, 1, ..., n-1\}$ for $n \in \mathbb{N}$.

Algorithm 1 Arithmetic Constant-weight Equality Operator

1: **algorithm** ARITH-CW-EQ(x, y)
2: $\quad h' = \sum_{i \in [\ell]} x_i \cdot y_i$
3: $\quad e = (1/h!) \prod_{i \in [h]} (h' - i)$
4: $\quad$ **return** $e \in \{0, 1\}$

3.3 Hashing Optimizations

We use a collection of hashing techniques to reduce the total number of comparisons and the cost of each comparison.

Hashing-to-Bins. Hashing-to-bins reduces the number of comparisons in a PSI protocol [20, 23, 32, 35]. Elements are distributed into a predetermined number of bins and comparisons are only performed between client and server elements in the same bin; since the bin in which an element is placed is determined using hashes of that element. When using this strategy, note that it is necessary to pad each bin with dummy elements up to a predetermined size to prevent the load of the bins from leaking information about the dataset.

There is a possibility that the number of elements in a bin exceeds the predetermined maximum, which would constitute a failure in the protocol. Thus, it is necessary to employ a strategy that cleverly manages elements that fall into the same bin.

Cuckoo hashing [31, 41] is a strategy for hashing elements into bins such that elements can be ejected from their current bin and placed in another bin after their initial placement. In Cuckoo hashing [31], there are m elements which are placed in $b = \varepsilon m$ bins, for some constant ε. Each bin holds at most one element and the placement of each element is determined using k hash functions, denoted as $H_i : \mathcal{X} \mapsto [b]$ for $i = 1, 2, \cdots, k$.

For each element x, we place x in bin $H_1(x)$. If bin $H_1(x)$ was already occupied by x', such that $H_1(x) = H_i(x')$, we evict x' from the bin. The new placement for x' is bin $H_{i+1 \mod k}(x')$. If that bin is empty, we are done; otherwise, we repeat this process, evicting the current resident of the bin. To ensure the protocol terminates, it is necessary to define an upper bound on the number of evictions permitted. If an empty bin is not found with the permitted number of evictions, the element is placed in a stash with a fixed size. Thus, to locate an element x, it is sufficient to look in bin $H_1(x)$, $H_2(x)$, $\cdots$, $H_k(x)$ or in the stash. When selecting parameters ε, k, and the stash size, please refer to past work analyzing the failure rates of Cuckoo hashing [3, 25, 31, 34].

Permutation-based Hashing Permutation-based hashing (PBH) is a technique to reduce the length of elements [3, 27, 32, 35]. We describe PBH in the following paragraph using the assumption that the number of bins is $b = 2^c$, for some $c \in \mathbb{N}$.

For a λ-bit element x, assume $x = x_H || x_L$ where $||$ is the concatenation operation and $|x_H| = c$ and $|x_L| = \lambda - c$. In the binning procedure, instead of placing x in bin $H(x)$, we place x_L in bin $b_x = x_H \oplus H(x_L)$. Now, if two elements x and y are placed in the same bin and match, this means that $x_L = y_L$ and $b_x = b_y$. From that, we have

$$b_x = b_y \Rightarrow x_H \oplus H(x_L) = y_H \oplus H(y_L) \Rightarrow x_H = y_H$$

Combining $x_L = y_L$ and $x_H = y_H$, we can deduce that $x = y$. Using this technique, we compare $\bar{\lambda}$-bit elements where $\bar{\lambda} = \lambda - c$. Since we can identify matches using only these shorter elements, we will only store elements of length $\bar{\lambda}$ in our bins. Thus, we term $\bar{\lambda}$ the *effective bitlength* and use it as the bitlength parameter throughout our descriptions and analysis of our protocol.

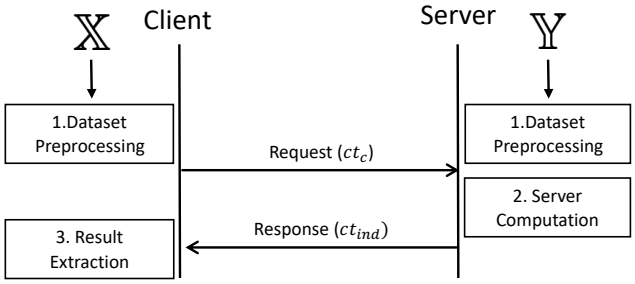

Figure 1: Stages of PEPSI.

4 PEPSI: Practically Efficient PSI

In this section, we describe our protocol PEPSI in detail; specifically, we describe the necessary steps to compute the intersection between a client set of size n and a server set of size m, where each set contains λ-bit elements. PEPSI is a PSI protocol that operates in one round and consists of three stages: Dataset Preprocessing, Server Computation, and Result Extraction. The client sends a *request message* to the server and the server responds with a *response message*. The request and response message constitute the total communication complexity of the protocol. At the end of the protocol, the client learns the intersection and the server learns nothing about the client set.

Figure 1 visualizes the steps of the protocol and the interaction between the client and server. In the base case (no variants), the Server Computation stage simply computes the intersection. This can be extended to other functionalities such as labelled PSI, computing a function on top of the intersection, and laconic PSI. See Section 5 for such variants. Table 2 summarizes all the notations used in the protocol description.

4.1 Dataset Preprocessing

In the first stage, the client and server must asynchronously preprocess their datasets to prepare them for the next stage. Preprocessing the dataset reduces the protocol latency when a client queries a server. This preprocessing is also helpful for the server, which has a large dataset and can reuse the preprocessed dataset for many queries by various clients. Both parties, client and server, preprocess their data in three steps: hashing optimization, encoding elements, and finally, batching (and encryption). Figure 2 visualizes the steps of the preprocessing. Note the actions in each step by a party are dependent on whether they are in the role of client or server. We

Table 2: Summary of notation for algorithm description.

Input Parameters	
$\mathbb{X}$	Client set
$\mathbb{Y}$	Server set
m	Client set size
n	Server set size
λ	Bitlength of elements
α	Maximum Error rate
Cryptographic Parameters	
N	Polynomial modulus degree
q	Coefficient modulus
t	Plaintext modulus
Binning Parameters	
b	Number of bins in T_c and T_s
γ	Client maximum bin load
μ	Server maximum bin load
$\bar{\lambda}$	Effective bitlength (used in PBH)
Constant-weight Code Parameters	
h	Hamming weight of constant-weight code
ℓ	Constant-weight code length
Auxiliary Notation	
T_c	Client table with b bins with γ elements
T_s	Server table with b bins with μ elements
T'_c	T_c with **encoded** elements
T'_s	T_s with **encoded** elements
pt_c	Plaintexts of clients elements
ct_c	Ciphertexts of clients elements
val_c	Ciphertexts of clients values
pt_s	Plaintexts of servers elements
val_s	Plaintexts of server values

provide a high-level overview here with relevant pseudocode in Appendix B.

Hashing Optimization. Our hashing optimization uses a hashing-to-bins strategy in conjunction with permutation-based hashing. Hashing-to-bins reduces the number of elements we need to compare, and permutation-based hashing reduces the size of elements that are inserted in the table (recall $\bar{\lambda} = \lambda - c$ as the effective bitlength from Section 3). The selected hashing-to-bins strategy in PEPSI must insert elements into b bins and does not require a stash. Elements that fall into the stash must be compared with all other elements, which severely diminishes the performance of our protocol. Moreover, permutation-based hashing cannot be used in conjunction with a stash. Hence, for the binning portion of the hashing optimization, we can use known hashing strategies that do not have a stash, such as Cuckoo hashing without a stash [33, 35] or Dual hashing [23].

The client and server have predetermined maximum bin loads, γ and μ, respectively, which are a function of their set sizes and the number of bins. After inserting elements into the bins, each party pad bins with dummy elements up to the maximum load of the bin. If more elements are inserted into a bin than the maximum load, the protocol fails. We denote the client's binning table with b bins and a maximum load of γ as T_c. Similarly, we denote the server's binning table with b bins and a maximum load of μ as T_s.

Encode to Constant-weight Codewords. The client encodes each $\bar{\lambda}$-bit element in T_c as a constant-weight codeword of length ℓ and Hamming weight h. The Hamming weight can be chosen freely but affects the communication-computation tradeoff (see Section 6). We set ℓ as a function of $\bar{\lambda}$ and h such that there is a unique representation for every $\bar{\lambda}$-bit element in T_c. So,

$$\ell = \ell(\bar{\lambda}, h) = \min\left\{\ell \in \mathbb{N} \mid \binom{\ell}{h} \geq 2^{\bar{\lambda}}\right\}. \tag{1}$$

We denote T'_c as the new table which holds the encoded elements. Dummy elements are encoded to the all-zero string of length ℓ. The server constructs T'_s from T_s using the same procedure as the client uses to construct T'_c from T_c.

Batching (and Encryption). In the last step, the client restructures T'_c to obtain a table of plaintexts. Recall that plaintexts are vectors of length N. Each position in $pt_c[i][j]$ corresponds to elements from one of the b bins. More concretely, $pt_c[i][*]$ contains the bits of the i-th encoded element in each of the b bins of T_c. We call this the client's i-th batch. Moreover, this process, which we call *batching*, allows the protocol to perform comparisons in different bins simultaneously. Each shade of red in Figure 2 shows the contents of one plaintext. The client additionally encrypts each plaintext with its secret key. The server constructs plaintexts pt_s in a similar fashion from the contents of T'_s and the server's i'-th batch is defined similarly. The server does not need to encrypt its plaintexts.

We assume $b = N$ in this example to simplify the explanation, but we show how to relax this assumption in relevant steps of the protocol.

4.2 Server Computation

The client sends the ciphertexts it has produced to the server for the next stage. In this stage, the server receives the ciphertexts of the client's elements and performs the set intersection (see Algorithm 2). At a high level, for each $i \in [\gamma]$ and $i' \in [\mu]$, the server compares batch i of the client's dataset with batch i' of the server's dataset. The output of this step is a table of ciphertexts of size $\gamma \times \mu$, which we denote by ct_{eq}. For $i \in [\gamma]$ and $i' \in [\mu]$, $ct_{eq}[i][i']$ denotes the result of comparing elements from batch i from the client and batch i' from the server.

Recall that the arithmetic constant-weight equality operator does not depend on the operands of the comparison. Hence,

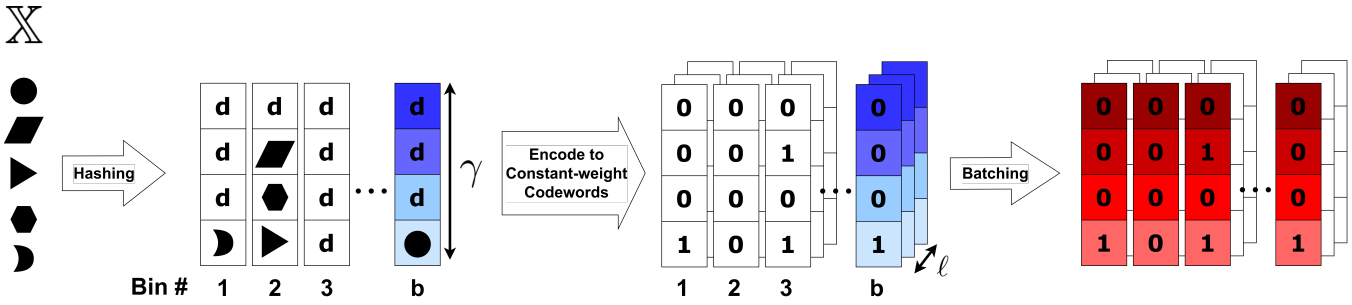

Figure 2: Client Dataset Preprocessing in PEPSI. d indicates dummy elements, b is the number of bins, and γ is the clients maximum bin load. From left to right: symbols represent the real-valued payload that is hashed into bins with a maximum bin load γ. The payload is encoded into bits using our constant-weight codewords, whereby the same color indicates the same payload, and finally encrypted into a ciphertext by batching across bins. The preprocessing outputs $\ell \cdot \gamma$ ciphertexts.

we can use it as a SIMD operator in line 4 of Algorithm 2.

Algorithm 2 Compute intersection of client and server set

1: **algorithm** INTERSECT(ct_c, pt_s)
2: **for each** client batch $i \in [\gamma]$ **do**
3: **for each** server batch $i' \in [\mu]$ **do**
4: $ct_{eq}[i][i'] \leftarrow$ ARITH-CW-EQ($ct_c[i], pt_s[i']$)
5: $ct_{ind}[i] \leftarrow \sum_{i' \in [\mu]} ct_{eq}[i][i']$
6: **return** ct_{ind}

If $b \neq N$, line 4 of Algorithm 2 is repeated $\lceil b/N \rceil$ times.

4.3 Result Extraction

The server sends the ciphertexts from the previous stage to the client for the final stage. In this final stage, the client decrypts the ciphertexts received from the server to obtain the indicator vector. The indicator vector specifies whether there was a match in the server set, for each element in the client set. From that, the client extracts the intersection. Algorithm 3 shows the procedure for this stage.

Algorithm 3 Extract intersection from server output

1: **algorithm** EXTRACTINTERSECTION
2: **for each** client batch $i \in [\gamma]$ **do**
3: $ind[i] \leftarrow \mathsf{Dec}(ct_{ind}[i], \mathtt{sk}_c)$
4: **for each** bin $k \in [b]$ **do**
5: **if** $ind[i][k] = 1$ **then**
6: Add $T_c[k][i]$ to intersection
7: **return** intersection

5 PEPSI Variants

Section 4 describes PEPSI for computing the intersection of two sets of λ-bit elements. However, the algorithm can be adapted for several other scenarios as well. Each adaptation adds extra steps to the protocol to achieve extra functionality or enhance performance. We describe these adaptations in detail in this section.

5.1 Optimization for Large Elements

In some applications, the client and server elements are very large, e.g., file names or 256-bit strings. In these cases, it is inefficient to compare large elements homomorphically. Instead, we map each element to a smaller λ-bit element using a hash function. However, if λ is too small, there may be colliding elements, which results in an incorrect result. Hence, we derive the failure rate of PEPSI if we map m client elements and n server elements to λ-bit elements. Conversely, we choose λ such that the failure rate of PEPSI is less than a given parameter α.

Not all collisions result in failure of the protocol, i.e., an incorrect result. Failure occurs when two unequal elements, one from the server and one from the client, have an identical mapping and are compared to each other by being placed in the same bin. Such an event produces an incorrect result since two unequal elements match. The upper bound on the failure rate is given in Lemma 1. Note that collision between two client elements (similarly, server elements) does not result in an incorrect result since client elements are not directly compared with each other.

Lemma 1. *When hashing m client elements and n server elements to λ-bit strings, the probability of failure in the protocol due to collisions is upper bounded by*

$$\frac{b\gamma\mu}{2^{\lambda}}, \tag{2}$$

where b, γ, and μ are the number of bins, maximum client bin size, and maximum server bin size, respectively.

We prove Lemma 1 in Appendix A. Using Lemma 1, we can see that, to bound the failure rate by $2^{-\alpha}$, we must

choose λ, such that $b\gamma\mu 2^{-\lambda} \leq 2^{-\alpha}$. Hence, we set $\lambda = \alpha + \lceil \log_2(b\gamma\mu) \rceil$.

5.2 Labelled PSI

PEPSI can be adapted to labelled PSI by altering the server computation and the result extraction. Algorithm 4 shows the adapted algorithm. In this algorithm, we denote the server labels as val_s, which is a vector of μ plaintexts. Moreover, the algorithm assumes that the label fits within one plaintext slot, but this can be extended to labels of arbitrary size. We show this extension in Appendix C and the corresponding result extraction procedure in Algorithm 4.

Algorithm 4 Server Computation and Result Extraction for Labelled PSI

1: **algorithm** LABELLEDPSI($ct_c[i]$,$pt_s[i']$)
2: **for each** client batch $i \in [\gamma]$ **do**
3: **for each** server batch $i' \in [\mu]$ **do**
4: $ct_{eq}[i][i'] \leftarrow$ ARITH-CW-EQ$(ct_c[i], pt_s[i'])$
5: $ct_{res}[i] \leftarrow \sum_{i'} val_s[i'] \cdot ct_{eq}[i][i']$
6: Send ct_{res} to the client
7: **algorithm** EXTRACTLABELS(ct_{res})
8: **for each** client batch $i \in [\gamma]$ **do**
9: $val[i] \leftarrow \mathsf{Dec}(ct_{res}[i], \mathtt{sk}_c)$
10: **for each** bin $k \in [b]$ **do**
11: **if** $val[i][k] \neq 0$ **then**
12: Add $val[i][k]$ to intersection labels
13: **return** intersection labels

5.3 Circuit PSI

PEPSI can additionally be extended to compute functions over the intersection. Such functions include PSI-Cardinality, PSI-Sum [23], PSI-Inner-Product [22, 28], and beyond. We introduce additional functionalities in the appendix.

PSI-Sum & PSI-Cardinality. In PSI-sum, the sum of server values associated with the elements in the intersection are output to the client. Algorithm 5 show the modified server computation and result extraction for PSI-sum, respectively. Kacsmar et al. use the same algorithm for securely computing PSI-sum and give proof of correctness and security [23]. PSI-Cardinality, which computes the size of the intersection, is a special case where the server values are equal to one.

PSI-Inner-Product. PSI-sum can be generalized to PSI-Inner-Product [28] as well. Assuming that the encrypted client values are denoted as val_c, we simply replace line 4 of Algorithm 5 with

$$sum \leftarrow \sum_{i' \in [\mu]} val_s[i'] \cdot \sum_i val_c[i] \cdot ct_{eq}[i][i'].$$

Algorithm 5 Server computation and result extraction for PSI-Sum

1: **algorithm** COMPUTEPSISUM(ct_c, pt_s)
2: **for each** client batch $i \in [\gamma]$ **do**
3: **for each** server batch $i' \in [\mu]$ **do**
4: $ct_{eq}[i][i'] \leftarrow$ ARITH-CW-EQ$(ct_c[i], pt_s[i'])$
5: $ct_{sum} \leftarrow \sum_{i' \in [\mu]} val_s[i'] \cdot \sum_i ct_{eq}[i][i']$
6: Sample vector $r \leftarrow [r_0, r_1, \cdots, r_{N-1}]$ s.t. $\sum r_i = 0$
7: $ct_{res} \leftarrow ct_{sum} + r$
8: **return** ct_{res}
9: **algorithm** EXTRACTPSISUM(ct_{res})
10: $res \leftarrow \mathsf{Dec}(ct_{res}, \mathtt{sk}_c)$
11: $ct_{sum} \leftarrow \sum_{k \in [b]} res[i]$
12: **return** ct_{sum}

6 Analysis of PEPSI

In this section, we provide an overview of the security analysis and details on communication and computation complexities. The communication and computation analysis is necessary to choose optimal parameters and enables PEPSI to achieve competitive performance compared to related work.

Security Analysis. The PEPSI protocol operates in the semi-honest model, i.e., the client and server follow the protocol but may try to infer extra information. The client input privacy is guaranteed due to the use of homomorphic encryption. The noise level in the homomorphic ciphertexts can reveal extra information about the server's dataset. Hence, we need a technique to make the noise level of the output ciphertexts indistinguishable from fresh ciphertexts. This is referred to as *circuit privacy* in the literature [5]. Techniques such as noise flooding [4, 9] achieve this property and can also be used in this work. Noise flooding is not implemented in the available homomorphic encryption libraries, so we do not perform that step in our implementation. However, it has a negligible effect on runtime, i.e., at the cost of one extra homomorphic addition, so our experimental results are unaffected. We choose all parameters such that we have 128-bit security.

Extending to the Malicious Model. The malicious security model removes the assumption that adversaries follow the protocol and considers active adversaries behaving arbitrarily. The malicious model for PSI has two limitations: 1) A malicious party may simply substitute its input and learn (parts of) the other party's set once the intersection is revealed, which cannot be prevented in the malicious model. 2) In a one-round protocol, arbitrary behaviour is only possible in the input-carrying first message. Hence, additional information can only be computed from the message (output) received in response to this first message. Using techniques such as input verification [6, 15] and verifiable homomorphic

encryption [40] we can augment our protocol to account for malicious clients and servers. However, we leave a details description of such a protocol for future work.

Notes on Performance Analysis. The communication and computation costs of PEPSI depend on many parameters, the most apparent of which are the set sizes and the bitlength of elements. Two parameters that can be tuned to optimize performance are the binning strategy and the Hamming weight.

The choice of binning strategy and associated parameters has been extensively discussed in previous works [31, 35], and using existing analysis, we can derive the relevant binning parameters b, γ, and μ, given the set sizes. Hence, in this work, we compute the communication and computation cost as a function of γ, μ, and b instead of the set sizes. As a result, our analysis is agnostic to the binning strategy.

The other parameter we can optimize over is the Hamming weight. The Hamming weight h is a parameter of the constant-weight equality operator. Cryptographic parameters depend solely on h as it determines the multiplicative depth. The code length is a function of the bitlength (and effective bitlength) and the Hamming weight, so it does not need to be optimized. In this section, we discuss how the communication and computation of PEPSI are affected by the Hamming weight and the tradeoff between these two metrics.

Analysis for Parameter Selection. Using the analysis from this section, we define strategies for optimally selecting parameters for PEPSI. There are two obstacles to optimizing the parameters in PEPSI. Firstly, there are circular dependencies between the parameters. For example, the cryptographic parameters, N and q dictate the number of bins, which influences the error rate which in turn influences the parameters of the constant-weight code. However, the Hamming weight of the constant-weight code determines the multiplicative depth, which puts a lower bound on the cryptographic parameters. The second obstacle is that there is no precise closed-form formula for the runtime so it requires experimental evaluation. Moreover, in many cases, there is no definitive choice for the parameters that optimize both communication and computation. In such cases, we visualize the communication computation tradeoff to assist in parameter selection. Hence, optimizing the parameters amounts to a non-trivial task that requires experimental evaluation and careful and systemic selection of the parameters.

6.1 Communication Complexity

The total communication complexity of PEPSI consists of the request and response messages. The response message depends on the function that is being evaluated. For example, if the set intersection is returned, the response is as big as the request message. In contrast, if we want the intersection cardinality, the response is only one ciphertext. In all the examples examined in this work, the response is smaller than the request size. Hence, to encompass all of these applications, we analyze and optimize the request size.

Asymptotic Communication Complexity. Theorem 1 derives the asymptotic complexity of the request in PEPSI as a function of the code length, ℓ.

Theorem 1. *The asymptotic complexity of the request message in PEPSI is $O(b \cdot \gamma \cdot \ell)$, where $\ell = \ell(\bar{\lambda}, h)$ as defined in Equation (1).*

Proof. As shown in Section 4, in the request, the encrypted plaintexts containing T'_c are sent to the server. T'_c is a table with dimension $b \times \gamma \times \ell$, with one bit in each cell of this table. Hence, the total communication is $O(b \cdot \gamma \cdot \ell)$. □

Corollary 1.1. *If we use Cuckoo hashing in PEPSI, then the communication complexity can be simplified to $O(m \cdot \ell)$ where m is the client set size and $\ell = \ell(\bar{\lambda}, h)$ as defined in Equation (1).*

Concrete Communication Cost. Theorem 1 only states the asymptotic complexity of the request size, but in practice, the size of the ciphertext must also be considered. Encoded elements in the client's set are put into $\lceil b/N \rceil \cdot \gamma \cdot \ell$ ciphertexts and each ciphertext in the request is approximately $N \cdot q$ bits, where q is the FV ciphertext modulus which is used. So overall, the size of the request in PEPSI is $\lceil b/N \rceil \cdot \gamma \cdot \ell \cdot N \cdot q$ bits (or simply $b \cdot \gamma \cdot \ell \cdot q$ when b is a multiple of N).

Note that the ciphertext modulus depends on the multiplicative depth, which depends only on the Hamming weight. N and q must also satisfy requirements for security which have been outlined in the literature [8]. For each Hamming weight, we select the smallest parameter set, which ensures the correctness of the result while also providing at least 128-bit security. Table 3 shows the polynomial modulus degree and bitlength of the coefficient modulus we use for each h.

Table 3: Polynomial modulus degree and coefficient modulus as a function of the Hamming weight.

h	1	2	3-4	5-8	9-16	17-32	33-64
$\log_2 N$	12	13	13	13	14	14	14
$\log_2 q$	72	144	168	204	240	276	312

Optimizing Communication Cost. The communication cost is derived as a function of the code length. However, the code length itself depends on the Hamming weight and effective bitlength, as shown in Equation (1). Hence, to minimize the communication complexity, we look for the Hamming weight which minimizes $\lceil b/N \rceil \cdot \gamma \cdot \ell \cdot N \cdot q$.

We know that γ does not depend on the Hamming weight so it acts as a constant in our optimization and can be removed.

So we can minimize $\lceil b/N \rceil \cdot \ell \cdot N \cdot q$, where $\ell = \ell(\bar{\lambda}, h)$. The effect of b is more convoluted and must be taken into account in the optimization. For simplicity, we assume $b \leq 4096$, which is less than the smallest possible value for N, so $\lceil b/N \rceil = 1$. We show results for other values of b in the appendix.

Figure 3 plots $\lceil b/N \rceil \cdot \ell \cdot N \cdot q$ with $b \leq 4096$ as a function of the Hamming weight for $\bar{\lambda} = 16, 32, 48$ to demonstrate that a minimum exists. The minimum communication occurs for $h = 8, 8, 23$, respectively.

We plot the optimal Hamming weight for communication for each effective bitlength in Figure 4 assuming $b \leq 4096$. In the case of a tie between two Hamming weights, we choose the smaller Hamming weight since it requires less computation. Generally, as the effective bitlength grows, a larger Hamming weight is required to achieve the smallest communication cost.

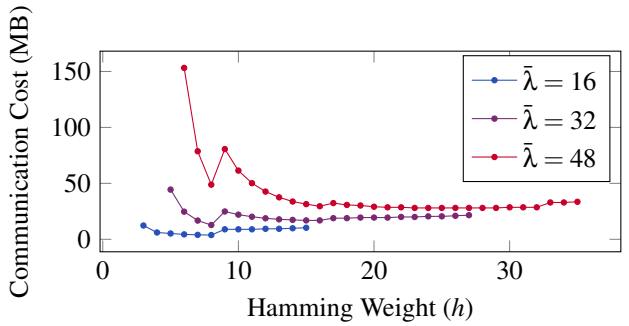

Figure 3: Code length as a function of the Hamming weight for $\bar{\lambda} \in \{16, 32, 48\}$ for $b \leq 4096$. The minimum occurs for a Hamming weight of 8, 8, and 23, respectively.

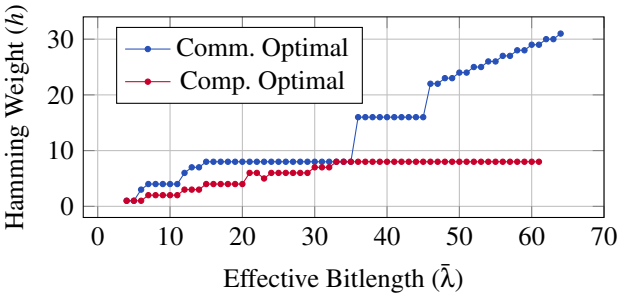

Figure 4: Hamming weight which optimizes communication and computation as a function of the effective bitlength ($\bar{\lambda}$) in blue and red, respectively. We assume $b \leq 4096$ in these graphs.

6.2 Computation Complexity

Amongst the stages of the protocol, the Result Extraction has a small computation overhead and also depends on the specific function that we derive. The data preparation can be done offline. Most of the online computation time is dedicated to the Server Computation stage. Within that stage, the majority of the runtime is dedicated to computing the encrypted indicator vector (Algorithm 2) and is required for any subsequent computation. The server computation algorithms described in Section 5.3 may perform additional operations after the indicator vector is derived, but in all cases, these operations are insignificant compared to calculating the indicator vector. Hence, we base our analysis and parameter selection on Algorithm 2 and calculate and optimize the runtime of this algorithm.

Approximate Server Computation Complexity. We approximate the computational complexity of the Server Computation stage by counting the number of expensive homomorphic operations, i.e., plaintext and homomorphic multiplications.

Theorem 2. *The number of homomorphic operations in Algorithm 2 is* $(\ell \cdot \mathtt{PM} + h \cdot \mathtt{M}) \cdot \lceil b/N \rceil \cdot \gamma \cdot \mu$ *where* PM *and* M *denote plaintext and homomorphic multiplication, respectively, and* $\ell = \ell(\bar{\lambda}, h)$.

Proof. The constant-weight equality operator in line 3 of Algorithm 2 consists of ℓ homomorphic multiplications and h plaintext multiplications [19]. In the general case when the number of bins is larger than N, line 4 of Algorithm 2 is performed $\lceil b/N \rceil$ times. Moreover, this line is repeated in two loops of length γ and μ, hence the total number of operations is $(\ell \cdot \mathtt{PM} + h \cdot \mathtt{M}) \cdot \lceil b/N \rceil \cdot \gamma \cdot \mu$. □

Optimizing Computation Cost. Given that we do not have a closed-form formula for the computation complexity of the homomorphic operators, optimizing the computation complexity is not possible. However, the concrete computation cost of PEPSI can be optimized empirically. Optimization is performed over the Hamming weight since we assumed that the bitlength, b, γ, and μ are given.

The client and server set sizes determine γ and μ, based on the binning strategy. The choice of h does not affect any of these parameters and only influences ℓ. Hence, γ and μ can be treated as constants in the optimization and it suffices to minimize $\left\lceil \frac{b}{N} \right\rceil \cdot (\ell \cdot \mathtt{PM} + h \cdot \mathtt{M})$. Theoretically, optimizing this term is difficult since there are no closed-form formulas for the runtime of homomorphic operations. Instead, we empirically optimize it by trying different Hamming weights. We set $\gamma = \mu = 1$ to eliminate the variance introduced by those parameters and run PEPSI using only one thread.

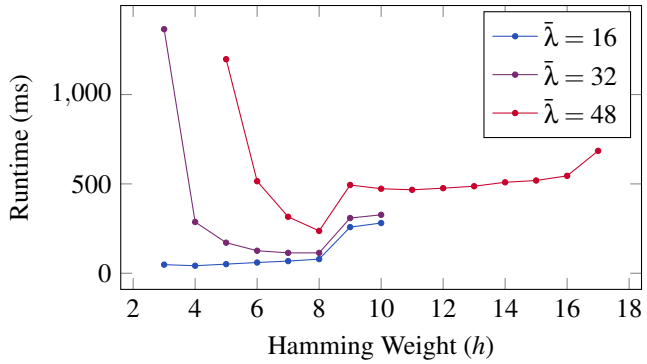

Figure 5: Runtime as a function of the Hamming weight for $\bar{\lambda} \in \{16, 32, 48\}$ for $b \leq 4096$. The minimum occurs for a Hamming weight of 4, 8, and 8, respectively.

Figure 4 shows, in red, the Hamming weights which optimize computation time for each effective bitlength assuming that $b \leq 4096$. Similar to the communication, the optimal Hamming weight for runtime increases as $\bar{\lambda}$ increases.

6.3 Communication/Computation Tradeoff

Note that choosing parameters that optimize the computation would increase communication costs and vice versa. In practice, it may be reasonable to choose parameters that fall somewhere between the two extremes. Figure 6 visualizes the communication/computation tradeoff for the intersection as the Hamming weight varies. We plot the communication/computation tradeoff for $\bar{\lambda} = 16, 24, 32$ assuming that $b \leq 4096$.

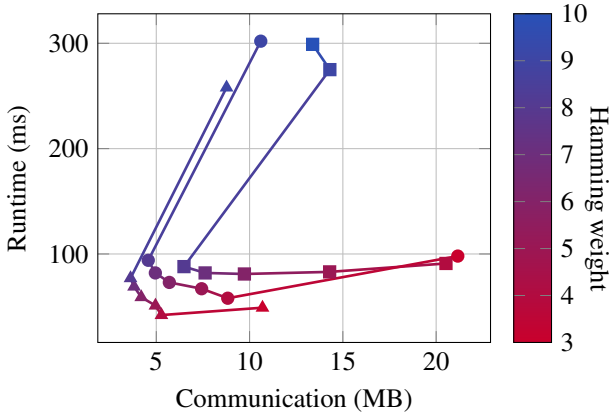

Figure 6: Visualizing the tradeoff between communication cost and runtime in PEPSI for $\bar{\lambda} = 16$ (triangles), $\bar{\lambda} = 24$ (squares), and $\bar{\lambda} = 32$ (circles) with $b \leq 4096$. Each point indicated running PEPSI with a specific Hamming weight which varies from 3 (on the lower right side) to 9 (on the upper left). We set $\gamma = \mu = 1$ in this example to isolate the effect of other parameters.

Notice that in some cases, the communication optimal may incur a high burden on the computation for very little gain, and vice versa. For example, in Figure 3 for $\bar{\lambda} = 48$, the communication cost decreases very slightly (less than 6%) as we increase h from 16 to 23. However, in Figure 5, the computation cost is much lower for $h = 16$ compared to $h = 23$ (over 40%).

7 Evaluation

Experimentation Details. We implement PEPSI in C++ using the Microsoft SEAL[1] library and measure runtime and communication. The SEAL library implements a variant of the FV cryptosystem and supports the operations we use (recall Section 3). Our implementation of PEPSI and the code used in our evaluation is available on Github [2]. We run experiments using two scenarios: 1) 32-bit elements and 2) large elements. 32-bit elements can be used to represent phone numbers in private contact discovery [23, 24]. Large elements are useful for applications such as checking for compromised credentials, where credentials could be strings of arbitrary length [29]. Parameters for these two scenarios are selected as follows:

Parameters for 32-bit Elements. Based on Figure 4, the best Hamming weight we can choose for the communication cost is $h = 8$, which results in $N = 2^{13}$ and $\log_2 q = 192$. As explained in Section 4, we round the number of bins up to a multiple of N for efficiency. Consequently, in our experiments, we have $b = N = 2^{13}$, $\bar{\lambda} = 19$, and $\ell = 24$. For our binning strategy, we use Cuckoo hashing with three hash functions without a stash, so $\gamma = 1$. We choose the parameters of Cuckoo hashing such that the failure rate is less than 2^{-40}. Based on the analysis in the literature [33, 35], the number of bins must be at least $1.27m$. The server bin size depends on the server set size and the number of bins. We use upper bounds for the server bin load found in the literature [31] and confirm the numbers experimentally using a simulation of the binning strategy.

Parameters for Large Elements. We choose the parameters such that the failure rate due to colliding elements is less than $\alpha = 2^{-40}$, so λ changes as the server set size increases. The binning strategy is Cuckoo hashing with three hash functions and no stash, so $\gamma = 1$. As a result, $b = N = 16384, \lambda = 40 + \lceil \log_2(b \cdot \gamma \cdot \mu) \rceil = 40 + \lceil \log_2(b \cdot \mu) \rceil, \bar{\lambda} = 40 + \lceil \log_2(\mu) \rceil$.

7.1 PSI Evaluation and Comparison

Comparison with DiPSI [23]. DiPSI [23] is the most similar work in the literature and operates in one round, similar to PEPSI. While PEPSI uses a homomorphic equality operator, similar to DiPSI, it improves on DiPSI in several aspects. The

[1] https://github.com/microsoft/SEAL
[2] https://github.com/RasoulAM/pepsi

two main differences are the use of a more efficient equality operator and permutation-based hashing. These differences result in significant improvements in concrete runtime and communication costs. Moreover, the advantage is increased by careful optimization of the parameters of PEPSI, as was mentioned in Section 6. Such optimizations are not required in DiPSI given that there is no variable parameter in the equality operator.

Table 1 shows that the asymptotic complexity of DiPSI and PEPSI is identical, but if we include parameters regarding the bitlength of elements, there is a difference. Table 4 displays the asymptotic complexity of the two protocols. The table also shows that the multiplicative depth of PEPSI does not depend on the bitlength, which results from our choice of equality operator.

Protocol	Communication	Computation	Mult. Depth
DiPSI	$O(m\lambda)$	$O(n\cdot\lambda)$	$\log_2\lambda$
PEPSI	$O\left(m\cdot\ell(\bar{\lambda},h)\right)$	$O\left(n\cdot\ell(\bar{\lambda},h)\right)$	$1+\log_2 h$

Table 4: Asymptotic complexity of DiPSI vs PEPSI in the unbalanced case, i.e., $m \ll n$.

All our experiments follow the unbalanced setting where the server has more elements than the client. DiPSI performs the intersection over 32-bit elements, so we do the same for a fair comparison. Cong et al. [14] allows to use elements of arbitrary length so we compare with the variant of PEPSI which can do the same. We measure all messages exchanged as the total communication and for computation, we measure the server's total runtime. We repeat experiments three times and report the average. However, the runtime of DiPSI is prohibitively high, so we take numbers from their paper [23].

Table 5 shows runtime of DiPSI and PEPSI for intersection of sets with 32-bit elements. This table shows that DiPSI is much slower than our work. PEPSI is at least four orders of magnitude faster than DiPSI. Most of this speedup can be attributed to the faster comparison operator, which we use in PEPSI. The communication of PEPSI is much smaller as well, which stems from 1) better parameters for the cryptosystem, 2) permutation-based hashing in conjunction with Cuckoo hashing, and 3) optimal Hamming weight, which minimizes communication. PEPSI requires 90% less communication compared to DiPSI. DiPSI can extend to labelled PSI and circuit PSI as well, which increases the total runtime and communication. We omit DiPSI from the rest of the evaluation due to its low performance.

To summarize, PEPSI is a strict improvement over DiPSI. It is non-interactive, similar to DiPSI, requires strictly less communication, is over three orders of magnitude faster, and is capable of all functionalities that DiPSI offers.

Table 5: Private set intersection over 32-bit elements using DiPSI and PEPSI. The bin size in PEPSI is $b = 8192$ in all cases. DNF indicates experiments which did not finish in less than an hour. * We copied the runtimes and communicating cost for DiPSI from their paper [23]. Note that the runtime of DiPSI is in minutes.

		DiPSI [23]		PEPSI				
n	m	Time (mins)	Comm. (MB)	μ	Time (s) Offline	Time (s) Online	Comm. (MB) Req.	Comm. (MB) Resp.
2^{20}	1024	600*	200*	526	0.11	0.83	4.1	0.11
	2048	350*	200*			0.81	4.1	0.11
	4096	190*	200*			1.5	8.1	0.22
2^{24}	1024	DNF	DNF	6710	1.5	8.9	4.1	0.11
	2048	DNF	DNF			9.0	4.1	0.11
	4096	DNF	DNF			17.5	8.1	0.22
2^{28}	1024	DNF	DNF	100565	19.9	133	4.1	0.11
	2048	DNF	DNF			133	4.1	0.22
	4096	DNF	DNF			260	8.1	0.22

Comparison with Cong et al. [14] Cong et al. [14] is the state-of-the-art amongst non-interactive PSI protocols with only two rounds of interaction. The work of Cong et al. has strictly improved over the work of Chen et al. [10, 11]. Cong et al. denote the time required to compute the OPRF of server elements as *offline* time. We report the offline and online time separately. We use the publicly available implementation of the work of Cong et al. [14], which is published on Github[3] and is called APSI. We use parameters provided with their code for each client and server set size pair. Table 6 compares PEPSI and the work of Cong et al. [14] for intersection of sets with large elements.

The communication cost of PEPSI is more than the work of Cong et al. in PSI and labelled PSI, but the gap narrows as the size of the labels increases. Regarding runtime, most of the server runtime (over 95%) in the work of Cong et al. [14] is spent computing a pseudo-random function for each server element in the offline phase. They state that this step could be computed offline and stored on the server. This is a valid assumption for some applications but does not apply to scenarios in which the server database frequently changes or when we require the overall runtime to be small. In the case of PSI (without labels), for smaller server set sizes, the protocol of Cong et al. has a better total runtime. However, for server sets with over 2^{24} elements, the total runtime of PEPSI is less than the work of Cong et al., with PEPSI requiring less time in the offline phase, but more time in the online phase. Hence, in the case of PSI, PEPSI is preferable for large server sets that frequently change.

7.2 Labelled PSI Evaluation and Comparison

We compare with the work of Cong et al., which is the state-of-the-art in non-interactive labelled PSI. They use the same

[3] https://github.com/microsoft/APSI

Label Size (Bytes)	n	m	Cong et al. [14] Time (s) Offline	Online	Comm. (MB) Request	Response	μ	PEPSI Time (s) Offline	Online	Comm. (MB) Request	Response
No Label	2^{20}	1024	5.6	0.44	1.3	1.2	296	0.11	7.8	30.0	0.22
		2048	5.7	0.82	2.7	1.5			7.1	30.0	0.22
		4096	6.1	0.94	3.5	2.2			7.0	30.0	0.22
	2^{24}	1024	99	1.2	1.8	2.1	3487	1.5	75.4	33.7	0.22
		2048	97	1.5	3.0	2.3			75.6	33.7	0.22
		4096	97	1.8	5.2	2.6			75.9	33.7	0.22
	2^{28}	1024	1770	7.3	3.5	9.5	50812	19.9	1144	39.2	0.22
		2048	1720	7.4	6.0	9.5			1154	39.2	0.22
		4096	1790	7.7	8.5	9.8			1141	39.2	0.22
32	2^{20}	256	92	2.4	4.2	3.8	296	0.11	7.2	30.7	2.9
		4096	95	2.4	4.4	3.5			7.0	30.7	2.9
	2^{22}	256	535	2.9	4.2	6.7	976	0.45	21.5	32.1	2.9
		4096	530	3.3	4.4	6.9			21.3	32.1	2.9
	2^{24}	256	DNF	DNF	DNF	DNF	3487	1.5	76.0	34.5	2.9
		4096	DNF	DNF	DNF	DNF			75.0	34.5	2.9
288	2^{20}	256	567	2.0	2.7	9.1	296	0.11	7.1	30.7	26.0
		4096	578	1.7	2.7	9.1			7.2	30.7	26.0
	2^{22}	256	3501	13.6	4.2	37.4	976	0.45	21.5	32.1	26.0
		4096	3388	14.2	4.4	35.3			21.5	32.1	26.0
	2^{24}	256	DNF	DNF	DNF	DNF	3487	1.5	76.5	34.5	26.0
		4096	DNF	DNF	DNF	DNF			76.1	34.5	26.0

Table 6: Private set intersection and Labelled PSI over large elements using the work of Cong et al. and PEPSI. The number of bins is set to $b = 16384$ in PEPSI for all cases.

parameters for the experiment but vary the label size. We choose 32-byte and 288-byte labels in our experiments. The results of this comparison are summarized in Table 6.

However, the work of Cong et al. has a high overhead when extending to labelled PSI instead of PSI. The computation cost of Cong et al. scales with the size of the label. In contrast, our work requires negligible additional server time to compute labelled PSI. The total runtime of PEPSI is consistently less than Cong et al. and the gap widens as the server set size increases, but PEPSI still requires more time in the online phase. So, in summary, in the case of labelled PSI, PEPSI is consistently faster than Cong et al., particularly for large, dynamic server sets and large labels but requires more communication.

7.3 Circuit PSI Evaluation and Comparison

Our evaluation of circuit PSI is focused on non-interactive solutions, which are compatible with the unbalanced setting. Hence, we omit solutions based on 2PC [33, 35], which require interaction between the client and server to compute the function. DiPSI can extend to circuit PSI and evaluate arbitrary functions but has a prohibitively high runtime, as we saw in Table 5. Recall that the work of Chen et al. [10, 11] and Cong et al. [14] cannot extend to circuit PSI. Instead, we compare with the work of Ion et al. [22], which is also a circuit PSI protocol but is limited to only a few functions: PSI-Cardinality and PSI-Sum-with-Cardinality. We use the public implementation[4] which is provided by the authors. PSI-Stats [42] performs the same operations as Ion et al. in the first two rounds of interaction and requires more operations when computing more complex functions. In the case of computing the PSI-Sum, PSI-Stats is identical to the work of Ion et al. Given that there is no publicly available implementation of PSI-Stats, we resort to the measurements from the work of Ion et al.

In our experiments in this section, we set the client set size to be $m \in \{1024, 2048, 4096, 8192\}$ and vary the server set size to observe the effect on communication and computation. The entire communication across all rounds is recorded and

[4] https://github.com/google/private-join-and-compute

n	m	Ion et al. [22] Time (s)	Ion et al. [22] Comm. (MB)	μ	PEPSI Time (s) Offline	PEPSI Time (s) Online	PEPSI Comm. (MB)
2^{18}	1024	52.9	20.0	100	0.01	3.1	27.6
	2048	71.9	20.7			2.7	27.6
	4096	64.8	21.9			2.7	27.6
	8192	74.7	24.4			2.8	27.6
2^{20}	1024	199	78.2	296	0.11	7.0	30.0
	2048	254	78.9			7.1	30.0
	4096	214	80.1			7.0	30.0
	8192	221	82.6			6.8	30.0
2^{22}	1024	950	311	976	0.44	21.4	31.4
	2048	797	312			21.4	31.4
	4096	791	313			21.4	31.4
	8192	808	315			21.5	31.4
2^{24}	1024	3153	1240	3487	1.5	75.8	33.7
	2048	3380	1240			76.3	33.7
	4096	3120	1250			75.8	33.7
	8192	3140	1250			75.6	33.6

Table 7: PSI-Sum using PEPSI and the work of Ion et al. [22]. Times are reported in seconds, and communication is in MegaBytes. DNF denotes instances which did not finish in under one hour. The number of bins is set to $b = 16384$ in PEPSI for all cases.

the runtime denotes the server runtime. Ion et al. [22] map elements to 256-bit strings using hash functions, so we use the variant of PEPSI with large elements to have a fair comparison. Table 7 summarizes the results of the experiments in this section.

Note that the cardinality is leaked whilst computing the sum in the work of Ion et al. [22]. In contrast, PEPSI does not have such leakage and is advantageous in this regard. Moreover, Ion et al. [22] is very limited in the functions that it can compute. Another observation is that the communication of the work of Ion et al. scales linearly with the server set size [22], whereas the communication cost of PEPSI increases at a much lower rate. For this reason, PEPSI is advantageous in the unbalanced setting and can scale to much larger server set sizes. The runtime of both protocols increases as the server size increases, but PEPSI is over 100x times faster than the work of Ion et al. [22]. One reason that PEPSI is faster than the work of Ion et al. [22] is the low per-element computation cost due to the use of batched homomorphic encryption. In contrast, the work of Ion et al. uses expensive modular exponentiations. The client must also perform expensive modular exponentiations, whereas in PEPSI, the client requires only a small amount of computation.

7.4 Summary of Evaluation

In our evaluation of PSI, we found that PEPSI demonstrates runtime that is comparable with other approaches [10, 14] but requires more communication. However, PEPSI excels particularly when the server size is very large and dynamic. In the context of labelled PSI, PEPSI is consistently faster than existing methods but requires more communication. The advantage of PEPSI increases as the label size increases. For circuit PSI, and specifically for PSI-Sum, PEPSI is faster than existing non-interactive approaches [22, 23], especially when dealing with larger server sets. Additionally, PEPSI has a significant advantage in communication cost, as it only depends on the client set size, making it highly efficient for large server sets. Furthermore, PEPSI offers the flexibility to easily extend to other functions, a capability that is not feasible with related work.

8 Conclusion

We propose PEPSI, a practical non-interactive circuit PSI protocol using homomorphic encryption. Some state-of-the-art PSI protocols cannot extend to circuit PSI. Those extending to circuit PSI require an interactive step with the client to compute the function or have the communication cost proportional to the server set size. These limitations are undesirable in the unbalanced setting. DiPSI, the only solution which does not have these limitations, has impractical runtimes. PEPSI addresses all these problems and proposes an efficient circuit PSI protocol with low communication overhead.

We use multiple techniques, such as constant-weight equality operators and permutation-based hashing that greatly improve the runtime and communication cost of PEPSI and result in a protocol that is competitive with existing work. We achieve competitive runtime and communication through careful optimization of parameters such as the Hamming weight.

PEPSI can compute the intersection of 1024 client elements with one million server elements in less than one second with less than 5 MB of communication. Functions such as sum and cardinality can be computed with negligible additional runtime. PEPSI is over four orders of magnitude faster than DiPSI, and 20x faster than the work of Ion et al. [22].

Acknowledgements

We gratefully acknowledge the support of NSERC for grants RGPIN-2023-03244, IRC-537591, the Government of Ontario and the Royal Bank of Canada for funding this research.

References

[1] Navid Alamati, Pedro Branco, Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, and Sihang Pu. Laconic private set intersection and applications. In *Theory of Cryptography: 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8–11, 2021, Proceedings, Part III*, pages 94–125. Springer-Verlag, 2021.

[2] Diego F. Aranha, Chuanwei Lin, Claudio Orlandi, and Mark Simkin. Laconic private set-intersection from

pairings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 111–124. Association for Computing Machinery, 2022.

[3] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 787–796, 2010.

[4] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*, pages 483–501. Springer, 2012.

[5] Florian Bourse, Rafaël Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In *Proceedings, Part II, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9815*, pages 62–89. Springer-Verlag, 2016.

[6] Jan Camenisch and Gregory M. Zaverucha. Private intersection of certified sets. In *13th International Conference Financial Cryptography and Data Security (FC)*, 2009.

[7] Nishanth Chandran, Divya Gupta, and Akash Shah. Circuit-PSI with Linear Complexity via Relaxed Batch OPPRF. In *22nd Privacy Enhancing Technologies Symposium (PETS 2022)*, jun 2022.

[8] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[9] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Multi-key homomorphic encryption from tfhe. In *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part II 25*, pages 446–472. Springer, 2019.

[10] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[11] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *23rd ACM Conference on Computer and Communications Security (CCS)*, 2017.

[12] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. pages 0–18, 1997.

[13] Michele Ciampi and Claudio Orlandi. Combining Private Set-Intersection with Secure Two-Party Computation. In *Security and Cryptography for Networks: 11th International Conference, SCN 2018, Amalfi, Italy, September 5–7, 2018, Proceedings*, pages 464–482, Berlin, Heidelberg, September 2018. Springer-Verlag.

[14] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication. In *27th ACM Conference on Computer and Communications Security (CCS)*, 2021.

[15] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *16th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2010.

[16] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling Private Contact Discovery. *Proceedings on Privacy Enhancing Technologies*, 2018:159–178, 10 2018.

[17] Thai Duong, Duong Hieu Phan, and Ni Trieu. Catalic: Delegated psi cardinality with applications to contact tracing. In *Advances in Cryptology – ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III*, page 870–899, Berlin, Heidelberg, 2020. Springer-Verlag.

[18] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[19] Rasoul Akhavan Mahdavi Florian Kerschbaum. Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[20] Michael J Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. Efficient set intersection with simulation-based security. *Journal of Cryptology*, 29(1):115–155, 2016.

[21] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *23rd International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2004.

[22] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 370–389, Genoa, Italy, September 2020. IEEE.

[23] Bailey Kacsmar, Basit Khurram, Nils Lukas, and et al. Differentially Private Two-Party Set Operations. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 390–404. IEEE, 2020.

[24] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile Private Contact Discovery at Scale. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1447–1464, Santa Clara, CA, August 2019. USENIX Association.

[25] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[26] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, CRYPTO'05, page 241–257, Berlin, Heidelberg, 2005. Springer-Verlag.

[27] Mikkel Lambæk. Breaking and fixing private set intersection protocols. Cryptology ePrint Archive, Paper 2016/665, 2016. https://eprint.iacr.org/2016/665.

[28] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. Private Join and Compute from PIR with Default. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 605–634, Cham, 2021. Springer International Publishing.

[29] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1387–1403, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Jack P. K. Ma and Sherman S. M. Chow. Secure-Computation-Friendly Private Set Intersection from Oblivious Compact Graph Evaluation. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, pages 1086–1097, New York, NY, USA, May 2022. Association for Computing Machinery.

[31] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, may 2004.

[32] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 515–530, Washington, D.C., August 2015. USENIX Association.

[33] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient Circuit-Based PSI with Linear Communication. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 122–153, Cham, 2019. Springer International Publishing.

[34] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III 37*, pages 125–157. Springer, 2018.

[35] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–35, 2018.

[36] L. Reichert, M. Pazelt, and B. Scheuermann. Circuit-based psi for covid-19 risk scoring. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 1–8, Los Alamitos, CA, USA, oct 2021. IEEE Computer Society.

[37] Jonathan Takeshita, Ryan Karl, Alamin Mohammed, Aaron Striegel, and Taeho Jung. Provably secure contact tracing with conditional private set intersection. In Joaquin Garcia-Alfaro, Shujun Li, Radha Poovendran, Hervé Debar, and Moti Yung, editors, *Security and Privacy in Communication Networks*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 352–373. Springer International Publishing, 2021.

[38] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, 2019.

[39] Ni Trieu, Kareem Shehata, P. Saxena, R. Shokri, and Dawn Xiaodong Song. Epione: Lightweight contact tracing with strong privacy. *ArXiv*, abs/2004.13293, 2020.

[40] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption, 2023.

[41] Udi Wieder. Hashing, load balancing and multiple choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3–4):275–379, 2017.

[42] Jason H. M. Ying, Shuwei Cao, Geong Sen Poh, Jia Xu, and Hoon Wei Lim. PSI-Stats: Private Set Intersection Protocols Supporting Secure Statistical Functions. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security*, pages 585–604, Cham, 2022. Springer International Publishing.

A Proof of Lemma 1

Lemma 1. *When hashing m client elements and n server elements to λ-bit strings, the probability of failure in the protocol due to collisions is upper bounded by*

$$\frac{b\gamma\mu}{2^{\lambda}}, \tag{2}$$

where b, γ, and μ are the number of bins, maximum client bin size, and maximum server bin size, respectively.

Proof. Let $\mathbb{P}[B]$ denote the probability of failure due to a collision. Let $\mathbb{P}[B_i]$ denote the probability of there existing a collision between any two of the elements of bin i, for $i \in \{1,2,\cdots,b\}$. So we have

$$\mathbb{P}[B] \leq \sum_{i\in[b]} \mathbb{P}[B_i]. \tag{3}$$

We know that, in a given bin, failure only occurs when two unequal elements, one from the server and the other from the client, have an equal constant-weight mapping. We know that the probability of a collision between two distinct elements from the domain is $2^{-\lambda}$, where the probability is in expectation over all perfect hash functions.

Hence, combining the facts stated above, we have that

$$\mathbb{P}[B] \leq \sum_{i\in[b]} \mathbb{P}[B_i] = \sum_{i\in[b]} \gamma\mu 2^{-\lambda} = b\gamma\mu 2^{-\lambda}. \tag{4}$$

□

Algorithm 6 Client and server data preparation

1: **algorithm** CLIENTDATAPREP($\mathbb{X}$)
2: Initialize T_c with b bins.
3: **for each** element $x \in \mathbb{X}$ **do**
4: Append x to the bins of T_c according to the binning strategy
5: Append dummy elements to fill each bin in T_c to the max
6: **for each** bin $k \in [b]$ **do**
7: **for each** batch $i \in [\gamma]$ **do**
8: $T'_c[k][i] = \text{CW-ENCODE}(T_c[k][i], \ell, h)$
9: **for each** batch $i \in [\gamma]$ **do**
10: **for each** bit $j \in [\ell]$ **do**
11: $pt[i][j] = [T'_c[1][i][j], T'_c[2][i][j], \cdots, T'_c[b][i][j]]$
12: $ct_c[i][j] = \mathsf{Enc}(pt_c[i][j], \mathtt{sk}_c)$
13: **return** ct_c

14: **algorithm** SERVERDATAPREP($\mathbb{Y}$)
15: Initialize T_s with b bins
16: **for each** server element $y \in \mathbb{Y}$ **do**
17: Append y to bins chosen by the binning strategy.
18: Append dummy elements to fill each bin in T_s to μ.
19: **for each** batch $k \in [b]$ **do**
20: **for each** batch $i \in [\mu]$ **do**
21: $T'_s[k][i] = \text{CW-ENCODE}(T_c[k][i], \ell, h)$
22: **for each** batch $i \in [\mu]$ **do**
23: **for each** bit $j \in [\ell]$ **do**
24: $pt_s[i][j] = [T'_s[1][i][j], T'_s[2][i][j], \cdots, T'_s[b][i][j]]$
25: **return** pt_s

B Data Preparation

The algorithm for server and client data preparation is provided as Algorithm 6. The client's secret key is denoted $\mathtt{sk}_c$, and is used for encryption. The encryption procedure is denoted with Enc. CW-ENCODE is the algorithm for mapping elements to constant-weight codewords. Precisely, we use the perfect mapping [19, Algorithm 3] and the lossy mapping [19, Algorithm 8] from the work of Mahdavi and Kerschbaum in the case of small and large elements, respectively. We refer the reader to the paper for the details of those algorithms.

C Large Labels

Algorithm 4 can be extended to the case where server labels are larger than one plaintext slot. The expensive step of the algorithm, which is matching client and server elements, does not need to be repeated. Instead, we simply repeat line 4 of Algorithm 4 for as many times that is required, depending on the size of the label.

D PSI-kth-Match.

Another function that the server can compute over the intersection is returning only one element in the intersection. This can be used to iteratively return the results of the protocol. For example, a data scientist may wish to see a sample of items in the intersection before deciding whether to recieve the entire intersection.

For simplicity, we only describe the cleartext algorithm for finding the kth match in a vector. The algorithm is designed such that it can be efficiently computed using homomorphic encryption. We include correctness and security proof to show that this algorithm can compute the kth match whilst not revealing any other information about the intersection. The full details on how to implement this algorithm using homomorphic encryption and the FV library are left for future work.

Assume that $I \in \{0,1\}^n$ denotes a vector with ones in some indices. The objective is to compute M such that $M[i_k] = 0$ where i_k is the position of the kth one in I. Algorithm 7 shows the algorithm that can achieve this.

Algorithm 7 Algorithm for kth match

1: **algorithm** ComputePSIkthMatch($I \in \{0,1\}^n, k$)
2: **for each** $i \in [n]$ **do**
3: $r \xleftarrow{\$} \mathbb{Z}_p$
4: $M[i] \leftarrow r \cdot (k \cdot I[i] - 1 - \sum_{i'<i} I[i'])$
5: **return** M

Theorem 3. *Algorithm 7 is correct, i.e., M is zero at the index of the kth one in I, and non-zero in all the indicies.*

Proof. Let $i_1 < i_2 < \cdots < i_s$ be the indicies such that $I[i_1] = I[i_2] = \cdots = I[i_s] = 1$ and $I[i] = 0$ for $i \neq i_j$. If $M[i] = k \cdot I[i] - 1 - \sum_{i' \leq i} I[i']$ then

$$M[i_k] = k \cdot I[i_k] - 1 - \sum_{i'<i_k} I[i'] = k - 1 - (k-1) = 0$$

Moreover,

$$j \neq k \Rightarrow M[i_j] = k \cdot I[i_j] - 1 - \sum_{i'<i_j} I[i'] = k - j \neq 0$$

$$i_j < i < i_{j+1} \Rightarrow M[i] = k \cdot I[i] - 1 - \sum_{i' \leq i} I[i'] = -1 - j \neq 0$$

which proves the theorem. □

The security of this algorithm is proven by showing that M reveals nothing about I other than the position of the kth one in the array.

Theorem 4. *If M is defined as in Algorithm 7, then M reveals nothing about I other than the index of the kth one.*

Proof. As shown in the correctness proof,

$$k \cdot I[i] - 1 - \sum_{i' \leq i} I[y'] \neq 0$$

for $i \neq i_k$. Moreover, due to the multiplication of r, which is uniformly random, $M[i]$ is also uniformly random, for $i \neq i_k$. Hence, $M[i']$ for $i' \neq i_k$ does not reveal any information about I. □

E Optimization for different values of b

Figure 5 showed the relationship between runtime and the Hamming weight for a fixed range of b. While there is not a close formula to show the effect of b, we can see in the graph below that for a larger b, the effect is roughly the same.

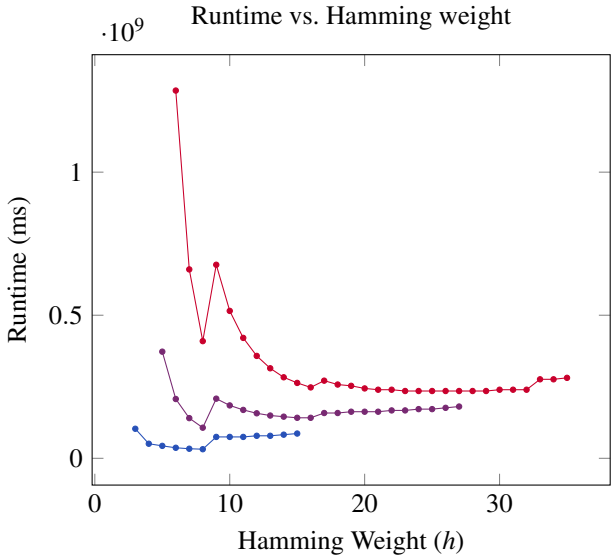

Figure 7: Code length as a function of the Hamming weight for $\bar{\lambda} \in \{16, 32, 48, 64\}$ for $4096 < b \leq 8192$. The minimum occurs for a Hamming weight of 8, 8, and 23, respectively.